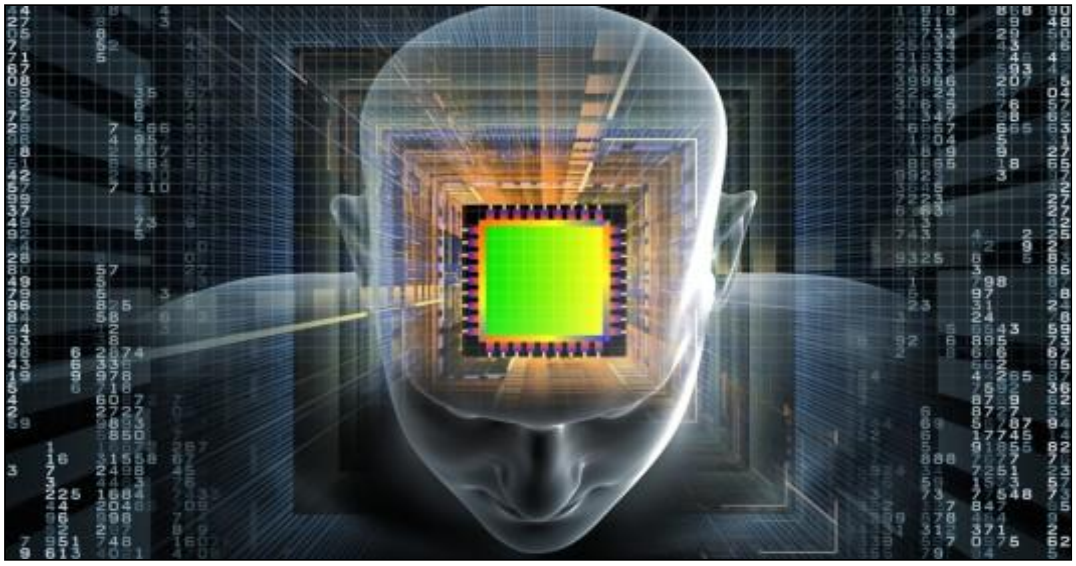
UNIVERSITY OF GOTHENBURG



Improving software comprehension process by Adoption of Cognitive Theories in large-scale complex software maintenance

An empirical research of cognitive theories in software maintenance

Bachelor of Science Thesis in the Programme Software Engineering&Management

Peter Chen

Xiaolei, Du

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2012

Improving software comprehension process by Adoption of Cognitive Theories in large-scale complex software maintenance
An empirical research of cognitive theories in software maintenance

Peter Chen
Xiaolei Du

Examiner: Helena Holmström Olsson

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover image resource:
http://teresaescrig.com/research-at-stanford-may-lead-to-computers-that-understand-humans/]

Department of Computer Science and Engineering
Göteborg, Sweden June 2012

Improving software comprehension process by Adoption of Cognitive Theories in large-scale complex software maintenance

An empirical research of cognitive theories in software maintenance

Peter Chen, Xiaolei Du

Department of Computer Science,

University of Gothenburg

Abstract

During the software maintenance process software comprehension is a time-consuming procedure. Fortunately, there are existing cognitive theories designed to improve software comprehension process. In this article, we intend to review six theories and perform an industrial case study in maintenance of a complex system. In order to find out whether to adopt cognitive theories in a specific maintenance task to improve the process of understanding the software or not, all six cognitive theories will be evaluated theoretically and one of them will be adopted in an industrial case study.

Keywords: *software comprehension, program understanding, cognitive theory, cognitive model*

1 Introduction

Software maintenance is an integral part of a software life cycle. ISO/IEC 14764 (2006), the international standard for software maintenance, defines software maintenance as one of the primary life cycle processes, and describes maintenance as the process of a software product undergoing "modification to code and associated documentation due to a problem or the need for improvement. The objective is to modify the existing software product while preserving its integrity." Software maintenance is an evolutionary development process. The term 'maintenance' relates to evolution and a continuance of development activities (D. Jin, 2005). As a kind of evolution, it inevitably companies with some issues and challenges in the software change process. One of the key challenges facing maintainers and maintenance efforts is comprehension of the system being maintained, that is, program comprehension or software understanding. Some activities involved in software maintenance, such as restructuring and reengineering, rely heavily on analysis and comprehension of the complex system structures and interactions that characterize both legacy and modern software systems (D. Jin, 2005). According to ISO/IEC 14764 (2006), program comprehension is defined in the category *Technical Issues*, and refers to how quickly a software engineer can understand where to make a change or a correction in a piece of software which this individual did not develop. Evidently, program comprehension is a major factor in providing effective software maintenance and enabling successful evolution of computer systems (A. V. Mayhauser et al, 1995). The importance of program comprehension for software maintenance is self-evident. Program comprehension is the essential part of software maintenance. The program comprehension process can be very time-consuming, and some estimate that up to 50% of the software maintenance effort is spent on understanding the software system at hand (S. Xu, 2005, W.J.Meng, 2006). In the real world, program comprehension is a challenge that software engineers face daily. Especially for organizations who bought their software from a third party, the maintenance of the software is always difficult. Therefore, the technicians of the organization need some strategies, like appropriate cognitive models and maintenance tools, to support their maintenance activities.

T. Reinikainen et al (2007) reveal that software comprehension is a human-intensive and typically task-driven activity. During the last few decades, lots of tools have been developed to support the software maintainers and analyzers to build a good understanding on the objective software system (T. Reinikainen et al, 2007). It is widely accepted that the tools that support software analysis and maintenance would go a long way towards addressing the constraints that software developers and maintainers work with on a day-to-day basis (D. Jin, 2005). A multitude of differences in program characteristics, programmer ability and software tasks have led to many diverse theories and research tools (M. A. Storey, 2005). Although program

comprehension tools share the common goal of simplifying the task of understanding large bodies of source code and building an appropriate representation of system structure, these tools differ at many levels: from their appearance to technical details to their philosophical approach (S. E. Sim et al, 2000). In actual software maintenance, the application of different theories, methods and tools will lead to many diverse results, which include different mental models of systems and different representations of system structures. According to the requirements of specific maintenance tasks and the maintainers' abilities, applying a cognitive model in program comprehension is possible to improve the efficiency of maintenance significantly. However, an ideal approach does not exist. Due to the fact that one cognitive model is not capable of solving all issues in software comprehension, hence, how to choose an appropriate cognitive model for a specific software maintenance task always challenges maintainers. That motivates us to research the adoption of cognitive models and tools that support large-scale complex maintenance tasks.

To direct our research, we identify two research questions: How does the program comprehension process affect the software maintenance process? How can a cognitive model or tool improve the software comprehension process in a large or complex software maintenance process? We will also adopt cognitive theories in a real industrial project, aiming at verifying the fact that program comprehension is a crucial factor of success or failure in software maintenance.
Aiming to answer the research questions, we will design an industrial case study. The industrial case study is a maintenance task for a driving simulator that involves two parties, SAFER (Vehicle and Traffic Safety Centre) and us. We will use literature reviews, observations and interviews as our data collection methods. During the research process, we intend to review three traditional and influential cognitive models, as well as three theories adopted in program comprehension. The three predominant theories of cognitive models are *Top-down*, *Bottom-up*, and *integrated meta-model* (M. P. O'Brien, 2003). These models have been identified and validated for more than 20 years and A. V. Mayrhauser et al suggests that applying them in software comprehension process could help software engineer to understand the source code (A. V. Mayrhauser et al, 1995). The three program comprehension theories proposed in the past 10 years. They are based on different theories and utilize various methodologies; hence, they have distinct application contexts, that is, they are not suitable for all maintenance tasks. Through the review of these theories, we will summarize the prominent characteristics of various cognitive theories, including three traditional cognitive models and three new fashion cognitive theories. We will observe the maintenance process before and after adopting a cognitive theory in software maintenance. Through interviewing the engineer involved in our research, we collect the opinions of practitioners for cognitive theory adopting. The data derived from our cognitive theory review, observation and interview is the evidence to support the claim that cognitive theories effectively improve software maintenance through improving program comprehension. The industrial case study we conducted reflects how the human factor influences the adoption of cognitive theory. Depending on these evidences, we will summarize some suggestions which should be useful to the latter maintainers when they are looking for tools supporting in cognitive process of software comprehension.

This paper is organized as follows. In SECTION 2, we introduce the theoretical framework built in the process of our literature review. Research from other authors about these six cognitive comprehension models and tools will be articulated in this section. In SECTION 3, we will describe our research approach to solve our research question, including research setting, research process, data collection, data analysis and limitations. In SECTION 4, we will present our research result from both the literature study and the empirical study. In SECTION 5, we will discuss the results for our empirical study based on the theoretical findings from literature and come up with some practical principles to maintainers. In SECTION 6, we will conclude our research and describe further research.

2 Theoretical frameworks

2.1 Program comprehension process and model

T. J. Biggerstaff et al (1993) defines program comprehension as: "A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program". In order to properly maintain a software system, maintainers have to fully comprehend this software they intend to maintain, or partially comprehend the software in case of specific maintenance task. If this knowledge is not readily available, they are

faced with the challenging task of gaining an understanding of the system's inner workings (S. G. M. Cornelissen, 2009). This process is known as program comprehension.

There are abundant cognitive models that have been developed to support program comprehension. M. P. O'Brien (2003) states that although these models differ significantly in their emphasis, they all consist of four common elements, namely, a knowledge base, a mental model, external representation, and some form of assimilation process. M. P. O'Brien (2003) also explicitly defines these components in his report. *External representations* are any 'external' views available in assisting the programmer when comprehending code, and are probably in form of software documentation, the source code itself, expert advice from other programmers familiar with the problem domain, or indeed, any other source code similar to the code under observation. *Knowledge base* can be defined as the programmer's accumulated knowledge before they attempt to understand the code and it will gradually expand in the comprehension process. The a*ssimilation process* is the actual strategy, which the programmer employs to comprehend the source code. A *Mental model* is a developer's mental representation of the program to be understood and describes a maintainer's current understanding of a software system. Program comprehension is typically referred to as the process involved in constructing an appropriate mental model of a software system to be maintained (B. Shneiderman, 1980, R. Brooks, 1983). Using the knowledge base, mental model, and external representations, the assimilation process continuously updates and augments the programmer's mental model (M. P. O'Brien, 2003).

Mental models are built and updated using actual strategies in the assimilation process, like adoption of cognitive models. The cognitive models are one of our emphases in this paper and they describe both the cognitive processes and the information structures needed to create a mental model (M. A. Storey, 2006).

2.2 A review of cognitive theories

In this section, we intend to review three cognitive models, *Top-down, Bottom-up, Integrated meta-model,* and three program comprehension theories created by authors in the recent 10 years; *Behavior-based model, Context-driven model and the Two-dimensional model.*

2.2.1 Cognitive models

Bottom-up, top-down, and the integrated model are the three major theories of program comprehension that try to model both the activities and the process involved in creating the mental models for comprehension tasks (W. J. Meng et al, 2006).

First and foremost, we introduce several concepts to assist us in understanding the models. *Plans* are knowledge elements for developing and validating expectations, interpretations, and inferences; they capture the comprehender's attention during the program understanding task (A. V. Mayrhauser et al, 1995). *Beacons* are recognizable, familiar features in the code that act as cues to the presence of certain structures (M. A. Storey, 2006). *Shallow reasoning* is a dynamic strategy in program comprehension. It does so without in-depth analysis and it has been adopted by many experts when they recognize familiar plans.

Top-down

Soloway and Ehrlich (1984) introduced a top-down model, and observed in their research that understanding in a top-down manner is appropriate when the practitioners are familiar with the source code or type of source code. Top-down understanding is typically adopted when the code or type of code is familiar. Theoretically, new code could be understood entirely in a top-down manner if the programmer had already mastered code that performed the same task and was structured in exactly the same way (A. V. Mayrhauser et al, 1995). A. V. Mayrhauser et al (1995) define top-down model is goal-oriented, in sense of the mental model contains a hierarchy of goals and plans. Rules of programming and beacons help decompose goals into plans and plans into lower level plans. Typically, shallow reasoning builds the connections between the hierarchical components. Brooks (1983) theorizes that hypotheses drive the cognition process in top-down model and the direction of further investigation. Understanding is complete when the mental model contains a complete hierarchy of hypotheses (A. V. Mayrhauser et al, 1995). A programmer first defines a hypothesis that describes the program, and then verifies it. Further hypotheses may be required in order to build up a hierarchy of hypotheses for verification. M. A. Storey (2006) defines top-down model that programmers understand a complete program in a top-down manner where the comprehension process is one of reconstructing knowledge about the domain of the program and mapping this knowledge to the source code.

Bottom-up

The bottom-up theory of program comprehension assumes that programmers first read code statements and then mentally chunk or group these statements into higher level abstractions. These abstractions (chunks) are aggregated further until a high-level understanding of the program is attained (B. Shneiderman et al, 1979).

Pennington (1987) suggests that programmers should build at least two mental models in the comprehension process. He found that when programmers or maintainers are unfamiliar with source code, they will build an elementary mental representation, called program model. This program model is a control-flow program abstraction and built from bottom up via beacons (A. V. Mayrhauser et al, 1995). After the program model is constructed, another model, situation model, is built from the bottom up, and based on the knowledge of real world domains, such as generic operating system structure and functionality for the operating system domain (A. V. Mayrhauser et al, 1995). The theory which interpret the program in a bottom up manner is labeled as bottom-up theory, in other words, understanding is built by reading the code then mentally chunking or grouping these lines of code into higher-level abstractions, (A. V. Mayrhauser et al, 1995, M. P. O'Brien, 2003). Letovsky et al (1986) also introduced the bottom-up theory, in which programmers gather together small chunks of source code in order to build up higher levels of abstraction, which are recursively grouped to produce a high level comprehension of a program (S. Xu, 2005).

Integrated meta-model

Von Mayrhauser and Vans (1993) observed that program comprehension is, in fact, neither a simple top-down nor a bottom-up process (S.C. Xu, 2005). A. V. Mayhauser and A. M. Vans (1995) developed a multilevel theory, which is known as the integrated model. This integrated meta-model evolved from the experiments carried out by von Mayrhauser and Vans, which concluded that programmers use a combination of assimilation processes when understanding software (M. P. O'Brien, 2003). They found in the experiment that, a combination of approaches becomes necessary for understanding large and/or complex systems. Therefore, the integrated model combines the top-down understanding of Soloway & Ehrlich (1984) with the bottom-up understanding of Pennington (1987). Pennington's bottom-up model consists of two sub-models, program model and situation model, which described in preceding contents. Pennington (1987) defines program model is programmers' first mental representation when code is completely new to them and it is a control-flow program abstraction. He also mentions the situation model is built based on knowledge of real world domain, such as generic operation system structure, and it would be completed once program goal is reached. Consequently, the programmer using integrated model actually switch among the three postulated areas or models (domain model, situation model, and program model) (S. Xu, 2005). Their integrated model consists of four major components: top-down, situation, program models and the knowledge base (A. V. Mayrhauser et al, 1995). The first three components describe the comprehension processes used to create mental representations at various levels of abstraction and the fourth component describes the knowledge base needed to perform a comprehension process (M. A. Storey, 2006). According to the familiarity of the source code and program application, maintainers can choose to invoke top-down model or bottom-up model as a starting point. M. A. Storey (2006) mentioned that when the code is familiar, top-down model can incorporate domain knowledge as a starting point for formulating hypotheses, otherwise, bottom-up model can be invoked and its program model serves as a control-flow abstraction. The situation model is the consequent when maintainers chose a bottom-up model and describes data-flow and functional abstractions. The knowledge represents the programmer's current knowledge and is used to store new and inferred knowledge, which support maintainers to build these three cognitive models (M. A. Storey, 2006).

2.2.1 New fashion theories

We will introduce three new fashion cognitive theories in this section. Comparing with traditional models, these theories have not been adopted in practice very common; however, they can be used to solve some specific problem in program comprehension relying on their predominant characteristics.

Behavioral IDE

Software maintainers are on their own in deciphering the dynamic behavior of the system, which is of primary concern in order to successfully understand the system and its design (R. Bayer et al, 2008). As R. Bayer and A. E. Milewski (2008) claimed, a possible solution to the problem of behavioral design feedback in IDEs is to center the design of an IDE on a cognitive model that represents a system in terms of its behavior instead of its structure, or in other words, create a behavioral IDE.

R. Bayer and A. E. Milewski introduced a prototype behavioral IDE that is capable to illustrate behavior design information in graphics and facilitates software maintainers more easily understand how a system works and locate relevant source code without documentations. This IDE named Dynamo is a Java-based
IDE that utilizes a behavioral representation of the system and this behavioral representation comes in the form of use cases and object interactions and sequenced events. Distinguishing with traditional way in which users interactive with source code through navigating the tree of files and packages within projects, R. Bayer and A. E. Milewski use sequence diagram in Dynamo IDE, which allows the user to navigate a software system via its behaviors, or use cases. They believe Sequence diagrams have been shown to be a highly efficient and quickly comprehended way to represent the behavioral view of a software system (R. Bayer et al, 2008). R. Bayer and A. E. Milewski stated that promoting the use of a mental strategy for system comprehension and problem solving is beneficial to the maintenance process, as it reduces wasted time searching through irrelevant source code. Consequently, they suggested the user of Dynamo should use a top-down cognitive model for solving maintenance tasks.

Context-driven process model

Current program comprehension research focuses mainly on developing better techniques and tools to tackle specific aspects of the comprehension problem, however, these techniques and tools are commonly not integrated with each other, due to a lack of integration standards or difficulties to share services among tools (W. J. Meng et al, 2006). It is result in maintainers do not know how these techniques and tools can collaboratively support a specific program comprehension task and face a specific comprehension task without any guidance. W. J. Meng et al (2006) are not only motivated by this need to synthesize these different information and knowledge resources utilized within a formal framework, but also to provide maintainers with a context during the program comprehension process itself. They introduce a formal process model that stresses an active approach to guide users (software maintainers and developers) to overcome this lack of context sensitivity while solving a comprehension task.

In their research, they utilize ontology to constitute the content of mental model. W. J. Meng et al (2006) claim that ontologies are often used as a formal explicit way of specifying the concepts and relationships in a domain of understanding. Another crucial element is Description Logic (DL), a knowledge representation formalism, which is used as a standard ontology language. W. J. Meng et al (2006) use ontologies and Description Logics to formally model the major information resources used in program comprehension and their interrelationships. In their model, ontological representation is used to model the information resources and the story-driven approach is used to model the interaction between users and the process context. In particular, W. J. Meng et al (2006) describe that the integration of resource representation and interaction must be supported by the structure and content of the ontological knowledge base.

Furthermore, W. J. Meng et al (2006) extend these models with an additional context sensitive support, a story driven approach. The story representation is an intuitive visual metaphor, and providing the maintainer with guidance on the use of different information resources to accomplish a particular task. W. J. Meng et al (2006) claim that story approach is capable to address three major issues, a) A metaphor that is familiar to users, b) A context that matches closely a comprehension process and therefore, can be used in actively guiding users while solving comprehension problems, and c) Stories can be expressed through different media, e.g. text, images, animation or other multi-media techniques.

Multi-dimensional cognitive model

S. Xu (2005) proposes a cognitive model for program comprehension which integrates constructivist theory and the Bloom's taxonomy of cognitive domain to form a two-dimensional model. There are six learning levels in Bloom's taxonomy of cognitive domain, Knowledge, Comprehension, Application, Analysis, Synthesis, and Evaluation. S. Xu (2005) described the constructivist learning theory as the learners actively and incrementally constructs their knowledge based on the preliminary knowledge. According to the existing theory, the two main activities are assimilation that describes how learners deal with new knowledge, and accommodation that shows how learners reorganize their existing knowledge. In order to describe assimilation and accommodation better, V. Rajlich and S. Xu (2003) subdivide these two activities as four processes, Positive assimilation and Negative assimilation, as well as Positive accommodation and Negative accommodation. In their future research, they named these four sub-processes

respectively as Absorption and Denial, as well as Reorganization and Expulsion.
The two-dimensional cognitive model consists of three components: Input, Cognitive process and Output. S. Xu (2005) defined that Input refers to the program to be understood or modified including the source code and documentation and programmers' existing knowledge and expertise, as well as Output contains the program with new functionalities, new documentation and new knowledge gained during the learning process. S. Xu (2005) defines cognitive process is composed of four activities at six Bloom learning levels, in other words, program comprehension is a learning process that enables the reconstruction of knowledge from program domain to design and task domain, with four cognitive activities at different learning levels.
S. Xu (2005) names this new model as a learning model due to the model stem from the existing constructivist learning theory and program comprehension itself is actually a learning process. He also states that this learning model emphasizes the importance of cognitive processes in developing their activities based on the existing program and the earlier knowledge of the programmers, which are fundamental in both knowledge and program performance.

3 Research method

This section describes the approach we took to conduct our research, as well as how the data was collected and analyzed. In the last part we illustrated limitations of this article.

3.1 Research background and setting

SAFER, Vehicle and Traffic Safety Centre at Chalmers is a joint research unit where 24 partners from the Swedish automotive industry, academia and authorities cooperate to make a center of excellence within the field of vehicle and traffic safety (http://www.chalmers.se/safer). In 2006, SAFER introduced STISIM Drive for car safety analysis. STISIM Drive, a fully interactive, PC-based driving simulator with unlimited customization potential, is ideal for a wide range of research and development applications concerning the driver, the vehicle, and the environment (road, traffic, pedestrians, visibility, etc.), drugs & pharmaceutical assessment and novice and professional driver training applications. Since the software was developed by an American company, and there is lack of Nordic virtual environments, thus SAFER simulator lab intends to implement some Nordic environment models as external libraries for STISIM Drive. SAFER purchased a software tool, Open Module Programming (OMP) that is able to construct external models for diverse environment visualization. The main goal of our task is to help SAFER implementing Nordic environments into their driving simulator. This maintenance task serves as the context of empirical study in our research. Our research is mainly conducted in the phase of maintenance planning and studies the adoption of program comprehension strategies and tools in context of complex software maintenance.

There are researches showing that software comprehension issues can lead to software maintenance slow down. (T. J. Biggerstaff et al., 1993; S. G. M. Cornelissen, 2009; M. P. O'Brien 2003; B. Shneiderman, 1980; R. Brooks, 1983; M. A. Storey, 2006). The research question of this paper is to find how can appropriate cognitive model or tool improve the software comprehension process in a large or complex software maintenance process? We approached this question from two perspectives: first of all, we reviewed literatures from previous research to find importance of software comprehension process and what causes the slowdown. Secondly, we performed an industrial case study together with an engineer from SAFER, and we interviewed the engineer after the case study. The results of the industrial case study are used to verify whether the theoretical solution can be adopted in practical problems.

3.2 Research Process

The research process consists of both a literature review and empirical research. In literature review, we found several papers about the importance of software comprehension during software maintenance. (T. J. Biggerstaff et al., 1993; S. G. M. Cornelissen, 2009; M. P. O'Brien 2003; B. Shneiderman, 1980; R. Brooks, 1983; M. A. Storey, 2006). Then we explored different existing cognitive models that can be applied to improve software comprehension problems. After we read through all the articles we found, we have identified six cognitive models that are relevant to our maintenance task. Moreover, the characteristics and capabilities of each model is analyzed to verify whether the cognitive model improve the software comprehension or not.

After the literature review, we started the industrial case study together with SAFER. During the industrial case study, the same maintenance task is given to the engineer from SAFER, they performed the maintenance task first time without introducing the cognitive model and then after some discussion and analysis they performed the maintenance task again with the cognitive model in mind, and we assisted and observed their performance. The

content of the maintenance task was to replace three current building models in the driving simulator with three Nordic style building models.

When they have finished the tasks first time, we get together and discuss the difficulties and the problems that occurred during the maintenance process. After all the feedback of results and experience were gathered, we reviewed six cognitive models together with SAFER, evaluated the cognitive model that is most suitable to solve the difficulties and problems during the maintenance process. The second time, we helped SAFER to operate the same maintenance task again, this time we applied suitable cognitive model; we guided and participated in the maintenance process. During the process, relevant data are recorded while performing the maintenance tasks together with SAFER's engineer. After they have finished the task second time, we gathered all data related to the changes in behaviors between first time and second time in terms of maintenance performance. Finally, we interviewed the engineers who participate in the industrial case study, several questions have been asked related to their experiences before and after adopting cognitive model.

3.3 Data Collection

The information gathered from literature review are collected through research papers related to cognitive models, and the data for industrial case study are collected through observation and interview of SAFER's engineer. Techniques we used to collect literature data are key words search using search engines, such as SpringerLink, IEEE Xplore, Elsevier and ACM. We tried to collect and read parts that are related to our topic. The data from empirical research were collected through observation and experience gained during maintenance task performance before and after the introduction of the cognitive model.

3.4 Data Analysis

3.4.1 Literature review

When collecting research papers related to importance of comprehension and cognitive model, we used key words such as software comprehension, cognitive model, improve software maintenance etc. The opinions from all collected research papers are used to discuss whether cognitive model improve software comprehension process or not.

3.4.2 Industrial case study

The data from industrial case study was analyzed by measuring the time taken of the maintenance task before and after we introduced the cognitive model to SAFER's engineer. To decide which model is most suitable for the maintenance task in SAFER, we analyzed the characteristics and capabilities of each model together with SAFER's engineers according to the difficulties and problems of the maintenance task. Later, we interviewed two of the SAFER's engineers, and asked their opinion of the differences before and after the introduction of cognitive model. All this information was used to discuss if the cognitive model could improve the software comprehension process in large-scale complex software maintenance.

3.5 Research limitation

The main limitation in our research is resource limitation, including time and human resource. Because SAFER bought their driving simulator from a third party, there are lack of technicians we can interview with, and the time to do STISIM Drive maintenance task is just about two and half month. There is limited time budget and human resource during the research process, thus we cannot perform any experiment to verify whether adopting cognitive in software maintenance is more effective than maintain without cognitive model. Moreover, STISIM Drive has many limitations for extensibility of the software, e.g. the building models in the software are encrypted by a third party.

4 Data

This section shows some empirical data collected during the industrial case study.

4.1 Interview Data

As we have described in research process section, we performed an interview with engineers from SAFER. A one-to-one interview was performed and involved the main maintainer of SAFER's simulation lab.

Through the interview, we found out that programming skill is not the most important issue we concerned in our industrial case study. The main maintainer from SAFER has basic knowledge of the programming languages used in driving simulator maintenance. In the case of lacking of programming skill, the maintenance process indeed improved after adopting cognitive model.

While SAFER was performing the maintenance task at first time without any guidelines, the task became very difficult to carry out. The comment from main maintainer: "We were totally lost, and don't know where to start with. The structure of the software is quite complicated, and the user manual is very time-consuming to

read. Thus, it was impossible for us to complete the task on time."

The major issue in the maintenance task is lack of understanding of software structure. Thus, the main maintainer states that it was difficult for them to figure out what should be changed, and how to change it without a good understanding of software structure. If they entirely understand the structure of the software, the maintenance task would be much easier for them to carry out.

Finally, we found that cognitive model is indeed helpful for SAFER's engineers, because after we introduced cognitive model to them, they understood the software structure much better than before, and figured out how to complete the maintenance task. The main maintainer said: "I think the model helped us understanding the structure of the driving simulator better. After you guys presented the cognitive model, we kind of understood where module we should make a change and which file should be override by new file. Besides, we had some experiences gained from the first time; hence, the maintenance task became much easier for us."

4.2 Cognitive Model Analysis Data

Hypothesis-driven model is one form of top-down cognitive model, which is a mature and verified theory for program comprehension. Maintainers normally select top-down model since they are part of familiar with the source code. In our case, we began with top-down model and developed using an as-need strategy (M. A. Storey, 2005). As-needed strategy refers to the programmer only focuses on the code sections related to the specific task at hand and does not study the dynamic relationships in much detail at all (M. P. O'Brien, 2003). STISIM Driving simulator is a complex and huge system, but we just focus on environment visualization and model building. In accordance with our needs, we established some goals in our maintenance task, and searched for the relevant modules to support these goals. Hypothesis is the main clue to guide conjecture of sub-goals and to build a hierarchy of goals. Through verifying of hypotheses and refining of goals, we had a goals hierarchy in hand (see Figure 1). It indicated the completion of goals hierarchy that every sub-goal is supported by one or more beacons. The sections of code would be reorganized to serve as beacons in the model. In realization of sub-goals and accumulation of reorganize beacons, domain model and program model would be built gradually. After analyzing and refining the goals, we identified code sections or functions that serve as beacons to support for corresponding sub-goals (see Table 1).

The main problem of SAFER's maintenance task is they spend too much time in software comprehension process, because the engineer from SAFER don't know the software structure of their driving simulator. The solution to improve software comprehension in SAFER's driving simulator is adopting cognitive model in comprehension process. SAFER's driving simulator, is programmed in Visual Basic and its models are constructed in C++. Because we are familiar with the semantic and syntax of Visual Basic and C++, thus after literature review, we have identified hypothesis-driven top-down model (HDTD model) proposed by Brooks (1983) can be adopted to improve program comprehension issue at SAFER.

Top-down strategy served as dynamic process strategy and comprehensive manner in the research. Our maintenance is an extension and complement of main functionality and can be defined as a perfective maintenance. After analyzed the comprehend task, we attempted to build a high level structure model as software comprehension strategy in STISIM Driving maintenance eventually.

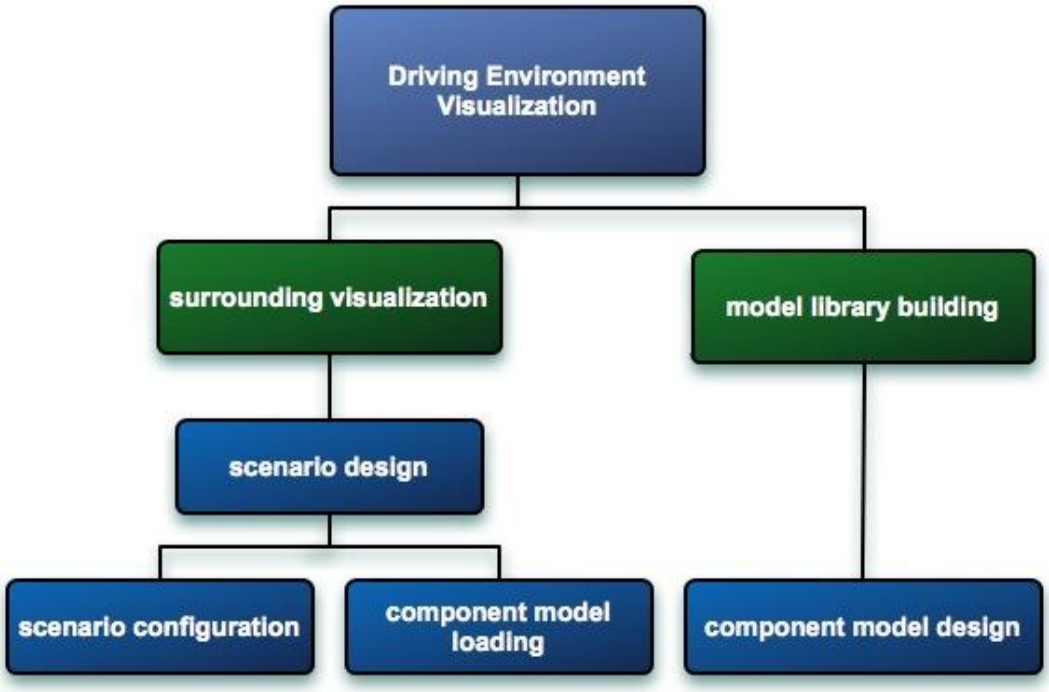


Figure 1 Goals Hierarch

In term of sub-goals and beacons, we had an external library development. This development is an abstraction of STISIM Drive and it is designed to display the relevant features and characteristics of system that we studied, and modified. Framing as a software comprehension model, this representation serves as the mental model and the process described above is the assimilation process. The external representation is a low level structure of STISIM Drive (see Figure 2) provided in product development documents. Together with existing knowledge base, we managed to create Nordic building models and adopt them to SAFER's driving simulator.

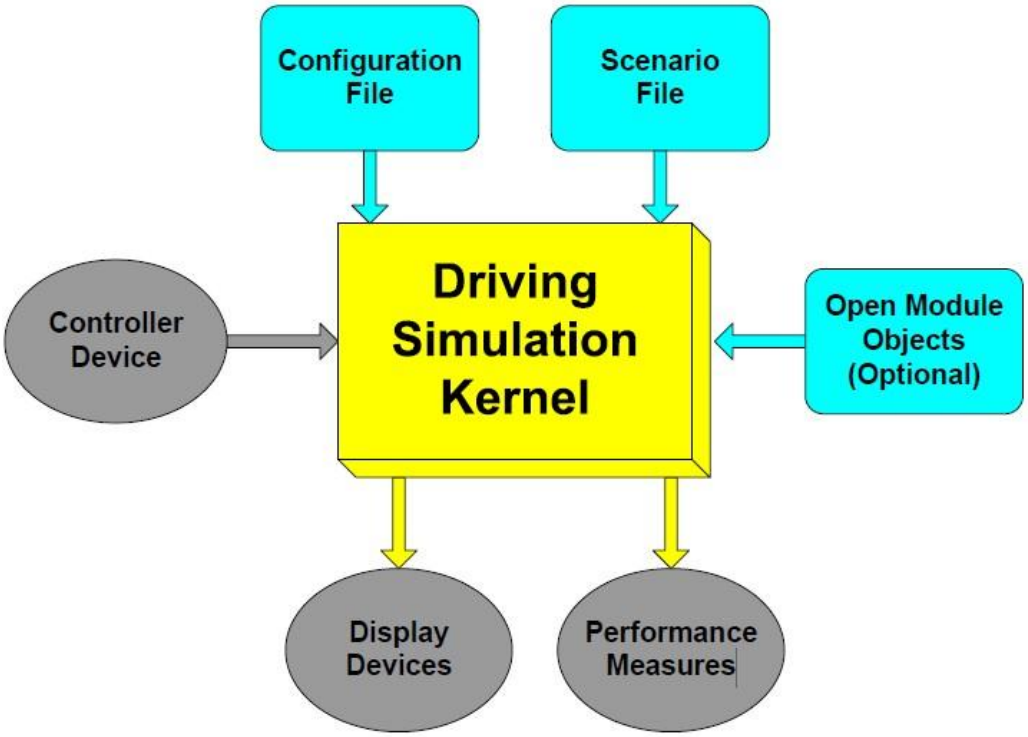


Figure 2 Low level structure of STISIM Drive

Goals(from high to low)				Beacons
Driving Environment Visualization	Surrounding visualization	Scenario design	Scenario configuration	Scenario Definition Language(SDL) programming
			Component models loading	**Dim Tools As New TJRWinToolsCls**(Create an instance of the graphics object) **Dim Graphics As New TJR3DGraphics**(Create an instance of the terrain object)
	Model library building		Component models design	AddNew, ControlInputs, Dynamics, Handle Crash…

Table 1 Sub-goals and corresponding beacon

5 Discussion

In this section, we will discuss our research focus in two different perspectives, which are also the questions directed the research. The arguments supported our discussion are the evidences gained from theoretical review and the data collected from industrial case study.

5.1 How does the program comprehension process affect the software maintenance process?

Theoretically, the significance of program comprehension for software maintenance is self-evident. We admit that the degree of program comprehension, to great extent, determines the quality, even success of software maintenance. That is, where to make the changes and how to make the changes depends on how well the software maintainers comprehend the software. Program comprehension is a core activity in software maintenance. If a program is not comprehended well, it will seriously impede the process of the maintenance project, which involves third-party or external maintainers, and obviously this will lead to some negative results. The most direct consequence is the growth of the maintenance life cycle. Additionally the software performance and stability might be reducing. In our research, we observed how SAFER's engineers maintained the STISIM Drive simulator. Their maintenance process is very struggling and time-consuming. A main reason is SAFER's engineers were not aware of their problem in program comprehension.

Theoretically, there exist various aspects affecting program comprehension, making it an inherently complex and difficult problem to address. W. J. Meng et al (2006) identify some of the major issues that will markedly affect the comprehension process. They include: the user's comprehension ability; the characteristics of the software system to be comprehended; the comprehension task to be performed; the tools and software artifacts (e.g. source code, documentation) available to support the comprehension process. Software artifacts include source code and all documentations. As dealing with source code involves a mental mapping between the system's code and its behavior, large amounts of source code are difficult to interpret directly because they result in a cognitive overload on the part of the maintainer. As a consequence, program comprehension is a rather time-consuming activity: research indicates that some 40% to 60% of the maintenance effort is devoted to understanding the software to be modified (SWEBOK, 2004). In SAFER, the same thing happened. Those four aspects affect their comprehension process. Firstly, SAFER's engineers did not have any successful experiences of maintaining this US simulator. Secondly, SAFER is just an end user of STISIM Drive simulator, in another word, they do use it but do not understand it very well. They did not have relevant knowledge background of this product, such as what kind of software to be maintained and to be comprehended, and software characteristics represent the software's application domain, size and complexity, programming language and architecture, and so on. Thirdly, in our observation, we found out SAFER's maintainers had a big problem in program comprehension. Thus, they did not know how to perform the maintenance task. Finally, SAFER is just a user so they did not need to cope with any software artifacts except the user manual. Unsurprisingly, SAFER's maintainers are stuck in software maintenance because of the bad comprehension of their product. That obviously reflects how important a good comprehension of maintained software for the achievement of success.

Program comprehension is a cognitive process and refers to activities human do understanding, conceptualizing, and reasoning about software. In this regard, a crucial aim of tools for software comprehension is to assist and improve human

thinking processes. Simply put, software comprehension tool are considered "good" if they support human cognition (A. Walenstein, 2002). We take it for granted that maintainers seek supporting to cognitive tools in the comprehension process of software maintenance.

5.2 How can a cognitive model or tool improve the software comprehension process in a large or complex software maintenance process?

5.2.1 Literature Review Findings

We conducted a theoretical analysis on various cognitive models in our industrial case study and aimed on finding an appropriate solution for our specific task. A. V. Mayrhauser et al (1995) proposed three models of evaluation criteria, s*tatic structures incorporating, dynamic process representation,* and *experimental validation degree*. They said that a static structure incorporating refers to "does the model incorporate static structures that represent persistent knowledge and the system's current mental representation?" Dynamic process representation refers to "does the model represent dynamic processes that build the mental representation using knowledge?" The last one, experimental validation degree refers to "the extent each model validated by experiments."

Improving the software comprehension process in traditional ways

We detected the mapping way of top-down model is from problem domain to programming domain or from strategic plan to implementation plan. The intermediate domain is the tactic plan. The dynamic process is only one direction, from top to bottom. The emphases of top-down models differ from one form to another. Brooks' model (1983) is the prototype of our model, and differs from other models in that all changes to the current system representation are driven by hypothesis (A. V. Mayrhauser et al, 1995). However, M. P. O'Brien presents the main limitation with this theory. It is that the model over-emphasizes the 'top-down' approach to comprehension, dismissing other strategies as 'degenerative processes'. It does not take into account, programmers who are inexperienced in the domain, who cannot use 'top-down' comprehension as they are lacking the knowledge to formulate the hypotheses in the first place. The knowledge base is always undefined in Brooks' cognitive model.

Integrated meta-model combines the top-down understanding of Soloway, Adelson, and Ehrlich3 with the bottom-up understanding of Pennington, hence, its dynamic process follows both top-down and bottom-up manner. As we mentioned in previous section, the integrated meta-model has four components, domain model, program model, situation model, and knowledge base. A. V. Mayrhauser et al (1995) said the knowledge base furnishes the process with information related to the comprehension task and stores any new and inferred knowledge. Other three component models may be active during the comprehension process and maintainers are able to switch between all three sub-models randomly. Top-down comes into effect predominately when the code is familiar. When the code is unfamiliar, maintainers can switch to bottom-up model. The most striking feature is self-evident, which is integrated meta-model supports frequent switching between top-down and bottom-up (M. P. O'Brien, 2003, M. A. Storey, 2005). M. P. O'Brien (2003) claims that the integrated meta-model has been used to identify the sequences of activities carried out to accomplish a comprehension goal and to understand how these are aggregated into higher-level processes. These can form the basis for identifying information needs during program comprehension and to define useful tool capabilities.

Comparing with top-down cognitive theory, the bottom-up model provides more details and describes the specific of cognition process and knowledge (A. V. Mayrhauser et al, 1995). Comprehension is built from the bottom up, and abstract concepts are formed by chunking together low-level information, accordingly, it is lack of higher level knowledge structure, such as design or application-domain knowledge (A. V. Mayrhauser et al, 1995, M. P. O'Brien, 2003). Pennington's model is a typical bottom-up cognition model. It contains mechanisms for abstraction. These mechanisms facilitate maintainers building the metal representation from control-flow abstraction to data-flow abstraction (M. A. Storey, 2005). As we mentioned before, Pennington suggest maintainers build at least two models in the comprehension process, program model and situation model. Control-flow abstraction of program, which captures the sequence of operations, is referred to construct a program model and is developed through chunking of microstructures in text (statement, control structures and relationships) into macrostructures (text structure abstractions) (M. A. Storey, 2005). A situation model is developed after the program model is fully assimilated. This model is a detailed representation of situation and helps maintainers understand a program, which includes knowledge about data-flow abstractions

and functional abstractions (M. P. O'Brien, 2003, M. A. Storey, 2005). However, O'Brien states that building this mental model is a time consuming effort, as it is constrained by the limited capacity of working memory.

New ways improve comprehension process

The behavioral IDE, Dynamo, is the foundation of a possible solution of program comprehension. It is capable of representing the design information of system in term of its behavior rather than its structure. Dynamo is developed by R. Bayer and A. E. Milewski (2008) and its main advantage claimed by R. Bayer et al is more easily and more quickly to gain a grasp of the software system they are maintaining, thus reducing time and cost of software maintenance. Dynamo facilitates maintainers to navigate a software system via its behaviors or use cases; hence, R. Bayer et al apply UML sequence diagrams to display the visual representation of behaviors and corresponding interactions between objects. This is a predominant characteristic of a behavior-based IDE. The reason stated by R. Bayer et al is that sequence diagrams have been shown to be a highly efficient and quickly comprehended way to represent the behavioral view of a software system. The features of Dynamo include zooming and scrolling. Most importantly, Dynamo is very interactive, since maintainers can easily shift between a behavioral representation of a system and its source code structure. One interesting point of view R. Bayer et al (2008) identified in their experiment is adopting an IDE with sequence diagram forces maintainers to use a strong and consistent strategy for program comprehension in software maintenance. Based on experimental results, R. Bayer et al suggest Dynamo users to use the top-down model in the maintenance process. The shortage of behavior-based cognitive solutions is evident like its strength. Even though Dynamo is active and flexible as R. Bayer et al (2008) described; it cannot illustrate structural information in higher level, like classes and functions. In addition, the study just focuses on software maintenance of simple systems, thus, utility of such approaches for more complex maintenance tasks or large-scale system should be explored (Bayer et al 2008).

W. J. Meng et al (2006) define their context-driven model as a formal process model to support the comprehension of software systems by using Ontology and Description Logic. The process itself is supported by two main components, the ontology manager and its query Interface and the story manager. They state their approach differs from existing work by providing a uniform ontological representation of the different information resources, including the context-sensitive user interaction with the comprehension process and the ability to reason across these knowledge resources. In other word, this ontological representation is a formal description that integrated all information resources and their interactions. The relevant information resources include *Task, User, Tools, Artifacts, and Software, Documents, and Historical data.* W. J. Meng et al (2006) summarize the competence of their context-driven process model in two aspects, serving as complementary to these ongoing tool integration efforts, and providing a formal ontological representation that supports reasoning across knowledge sources and provides context support and guidance during the comprehension process itself.

S. Xu's (2005) multi-dimensional cognitive model has two core theories, constructivist learning theory and Bloom's taxonomy of cognitive domain. Comparing with top-down model or bottom-up model, S. Xu (2005) claims that multi-dimensional model is more complete and detailed. It explains all the program comprehension processes by integrating both top-down and bottom-up models. It also classifies the cognitive activity during program comprehension into four activities, *absorption, denial, reorganization and expulsion* (S. Xu, 2005). In this way, maintainers are facilitated to get and to comprehend the knowledge so as to synthesize information and to generate hypotheses.

Summary of cognitive theories

The strengths and the drawbacks of diverse cognitive theories and models limit its adoption in program comprehension. Every theory or model has their own features and it is probably suitable for a kind of case or appropriate to cope with a sort of specific task. We concentrate on the theories and models, which are elaborated in Theoretical Framework, and their striking capabilities and limitations. We will analyze and summarize these cognitive theories.

Top-down cognitive model is driven by hypothesis, whereas, the mental representation could be changed or updated by other means – for instance, novice maintainers may resort to a bottom-up model because of hypotheses fail or they may attempt to a strategy-driven method, like opportunistic strategy (A. V. Mayrhauser et al, 1995, M. P. O'Brien, 2003). A. V. Mayrhauser et al (1995) concludes that both top-down and bottom-up use a matching process between what is already known (knowledge

structures) and the artifact under study, and no one model accounts for all behavior as programmers understand unfamiliar code. They also claim the integrated meta-model responds to the cognition needs for large software systems, accordingly, top-down and bottom-up are applicable for small scale code experiments and maintenance. It combines relevant portions of the other models and adds behaviors not found in them-for example, when a programmer switches between top-down and bottom-up code comprehension. Multi-dimensional cognitive model explains the cognitive activities in detail and it can also be applied in different cases (S. Xu, 2005). Other two cognitive theories, behavior-based model and context-base process model rely on the specific case or maintenance task much more. Von Mayrhauser and Vans (1998) claimed that, the models used may vary depending on the tasks and the programmers' command of knowledge on domains and programming, therefore, maintainers can adopt these two theories in accordance with needs and models' characteristics. Naturally, program comprehension is a goal-oriented and hypothesis-driven problem-solving process.

5.2.2 Industrial Case Study Findings

Interview findings

Through one to one interview with the main maintainer from SAFER, we find out that the cognitive model helped the software comprehension of the driving simulator in SAFER. However, we noticed some aspects through interview that might affect the result of adopt cognitive model in software maintenance.

First of all, we perform the maintenance task the first time without cognitive mode, and then perform the same task again after cognitive model has been introduced. This can affect the result, because at the second time, user has the experience of deal with the same task even without cognitive model.

Secondly, the programming skill of SAFER's engineer is quite basic. At the second time, we guide them to perform the maintenance task during the process, which can affect the result. Since we have better programming skills than SAFER's engineers.

Thirdly, the maintenance task in our industrial case study can be solved by adopting cognitive mode, but it might not be that easy to find appropriate cognitive model for every software maintenance task. Sometimes it requires much higher programming skills for the maintainer.

Finally, there are some limitations of our industrial case study findings, but through one to one interview with SAFER's main maintainer, we noticed that the cognitive model in this case definitely improved understanding of the software structure. This means, the cognitive indeed shorten the time consumption of software comprehension process.

Model analysis findings

W. J. Meng et al (2006) mentioned that the comprehension task to be performed is a major issue affecting program comprehension process. In our industrial study, we adopt top-down model into our maintenance task. According to A. V. Mayrhauser et al, new code could be understood entirely in a top-down manner if the programmer had already mastered code that performed the same task and was structured in exactly the same way. The goal of our maintenance task is to adopt Nordic environment into the current driving simulator, thus we sort out the structure of the software and looked into the current land terrain, building and traffic sign models. By study the mechanism of current models, our maintenance task become much easier, since we already understand how current models are build and structured, we can just create new models with Nordic environment by ourselves.

Supported by HDTD model, we partially comprehended the software rather than to fully understand the whole program. We successfully detected the place to be changed through using of goals hierarchy so that the maintenance time-consuming was reduced remarkably. The result of applying a cognitive model seems to have improved the software comprehension process a great deal. The time taken before introducing cognitive models to SAFER, took them many hours to achieve the result, but after we have introduced the cognitive model, the time taken of the same maintenance task become about half hour to one hour for each task.

Summary

In the maintenance task, we interviewed SAFER's engineer about their opinion of using cognitive models. The result of the interview seems that, they do think the cognitive model we introduced is quite helpful. Because, even though they do not understand much about programming, but the cognitive model helped them to understand the behavior pattern of the software itself, thus it is much easier to find the specific part of code and modify them. In terms of data collected from observation, participation, discussion and interview, we are able conclude that the comprehension process is improved by

adopting a cognitive model. The time-consuming on diagnosis and integration is reduced significantly, and the life-cycle of maintenance is shortened as well.

6 Conclusions and future work

Large-scale complex software maintenance process usually takes quite a lot of time, mainly due to the comprehension process. The goal of this paper is trying to show that adopting the relevant cognitive model as a guideline in the comprehension process could speed up the large-scale complex software maintenance process. Through the literature study, we found theoretical proof from several authors (T. J. Biggerstaff et al 1993, S. G. M. Cornelissen, 2009, M. P. O'Brien 2003, B. Shneiderman, 1980, R. Brooks, 1983, M. A. Storey, 2006) to show that software comprehension play a very important role in the software maintenance process; also, a cognitive model helps people understand the software structure and behavior better. The maintenance task in our case study is large-scale complex software maintenance. In this empirical study, we found that adopting the relevant cognitive model from the theoretical finding could shorten the large-scale complex maintenance task process.

These findings allowed us to give the following suggestion to maintainers of software:

- Always bear software comprehension in mind first while dealing with software maintenance issues.
- Read how others use cognitive models in software maintenance before the start of the maintenance process.
- Analyze different cognitive models and become familiar with them before planning the maintenance task.
- Find relevant cognitive models by analyzing the characteristics of the model and how well it suits the maintenance problem you have.
- In addition, we suggest maintainers adopt an as-need strategy in small scale or functional maintenance.

In the future, we would like to complete our evaluation in both a theoretical way and an empirical way and to conduct empirical research aiming to verify the effectiveness of cognitive theories. We believe this paper is a very useful reference for people who are experiencing comprehension problems in a software maintenance task. However, the comprehension issues vary from case to case, thus not all the comprehension issues can be solved with one cognitive model, and sometimes people need to define their own cognitive model to overcome issues that cannot be resolved with existing cognitive models.

Acknowledgement:
Sincere thanks for comments, feedback and encouragement from Helena Holmström Olsson and Björn Olsson while writing this paper.

REFERENCE

Bayer, R., Milewski, A. E., 2008. Improving Software Maintenance Efficiency With Behavior-Based Cognitive Models. IEEE International Conference on Systems, Man and Cybernetics.

Biggerstaff, T. J., Mitbander, B. G., Webster, D., 1993. The Concept Assignment Problem in Program Understanding. Proc. International Conference on Software Engineering.

Brooks, R., 1983. Towards a theory of the comprehension of computer programs. Int. J. of Man-Machine Studies, pp. 543-554.

Cornelissen, S. G. M., 2009. Evaluating Dynamic Analysis Techniques for Program Comprehension. pp. 13-15.

Guide to the Software Engineering Body of Knowledge, 2004, SWEBOK.

ISO/IEC 14764 IEEE Standard, 2006. Standard for Software Engineering - Software Life Cycle Processes - Maintenance, Software & Systems Engineering Standards. Committee of the IEEE Computer Society

Jin, D., 2005. Design Issues for Software Analysis and Maintenance Tools. 13th IEEE International Workshop on Software Technology and Engineering Practice. University of Manitoba.

Letovsky, S., Soloway, E., 1986. Delocalized plans and program comprehension. IEEE Software, 19(3), pp. 41 - 48.

Mayhauser, A. V., Vans, A. M., 1995. Program Comprehension During Software Maintenance and Evolution. IEEE Computer, pp. 44-55.

Mayrhauser, A. V., Vans, A. M., 1998. Program understanding behavior during adaptation of large scale software. 6th International Workshop on Program Comprehension.

Meng, W. J., Rilling, J., Zhang, Y. G., Witte, R., Mudur, S., Charland, P., 2006. A Context-Driven Software Comprehension Process Model. IEEE Workshop on Software Evolvability.

O'Brien, M. P., 2003. Software Comprehension – A Review & Research Direction. Technical Report, University of Limerick.

Pennington, N., 1987. Comprehension Strategies in Programming. Proc. Second Workshop Empirical Studies of Programmers, Ablex Publishing, pp. 100-112.

Rajlich, V., Xu, S., 2003. Analogy of Incremental Program Development and

Constructivist Learning. 2nd IEEE Int. Conf. On Cognitive Informatics, pp. 98 – 105.

Reinikainen, T., Hammouda, I., Laiho, J., Koskimies, K., Systä, T., 2007. Software Comprehension through Concern-based Queries. 15th IEEE International Conference on Program Comprehension, Tampere University of Technology & Nokia Research Center.

Shneiderman, B., 1980. Software Psychology: Human Factors in Computer and Information Systems. Winthrop Pub.

Shneiderman, B., Mayer R., 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, pp. 219-238.

Sim, S. E., Storey, M. A., 2000. A Structured Demonstration of Program Comprehension Tools. IEEE Computer, pp. 184-193.

Soloway, E., Ehrlich, K., 1984. Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, pp. 595-609.

Storey, M. A., 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. 13th International Workshop on Program Comprehension, University of Victoria.

Von Mayrhauser, A. V., Vans, A. M., 1993. From program comprehension to tool requirements for an industrial environment. 2nd Workshop on Program Comprehension, pp. 78-86.

Walenstein, A., 2002. Theory-based Analysis of Cognitive Support in Software Comprehension Tools. 10th International Workshop on Program Comprehension (IWPC'02).

Xu, S., 2005. A Cognitive Model for Program Comprehension. Third ACIS International Conference on Software Engineering Research, Management and Applications.